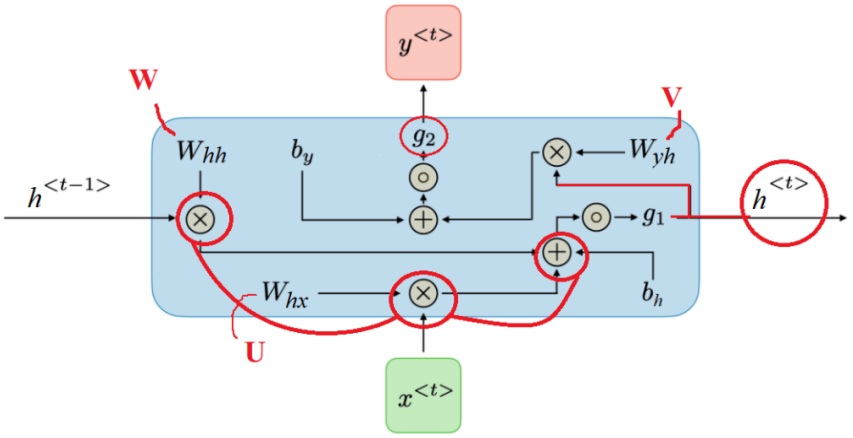
RNNs

Mình không intro quá dài nữa vì thật ra RNN đã khá quen thuộc rồi, mình notes lại những gì cần nhất để hiểu, đặc biệt là phần vì sao RNN ăn vanishing gradient, giải thích sự ra đời của LSTM để khắc phục điều đó.

1. *RNN (Recurrent Neural Network)*

RNN Recurrent Neuron Network là một mạng neuron thường được sử dụng trong các bài toán xử lý chuỗi như chuỗi time, chuỗi ký tự (nlp),... Lý do nó làm được điều này vì nó có cơ chế lưu trữ 1 cái hidden state (trạng thái ẩn, ký hiệu là h), cái này nó sẽ lưu lại thông tin của bước trước. Nó như kiểu bộ nhớ tạm thời ấy, hiểu đơn giản là vậy.

Đây là những gì diễn ra trong 1 step RNN được mô tả bằng hình:



- Hình dung đơn giản: Whh, Whx, Wyh là ma trận trọng số cần train, nói chung nó ứng với h, x và y.

  Mô tả công thức đúng theo như cái hình kia sẽ là: ở mỗi step sẽ có 2 đầu ra, một cái h và 1 cái y:

  Trạng thái hiện tại:

$$h^{<t>} = g_1(W_{hx} \cdot x^{<t>} + W_{hh} \cdot h^{<t-1>} + b_h)$$

Dự đoán ra yt:

$$y^{<t>} = g_2(W_{yh} \cdot h^{<t>} + b_y)$$

g1, g2 là các hàm kích hoạt, g1 thường là Tanh/ReLU, g2 là một hàm kích hoạt nào đó tùy theo yêu cầu đề bài, thường cũng là softmax hay sigmoid nếu cho mấy bài phân loại, hoặc có thể nó không là 1 hàm nào luôn, chỉ là viết vào quy ước chung vậy thôi.

Từ hình và công thức ta có thể hiểu đơn giản kiểu work của RNN:

- Nó lưu trữ thông tin của trạng thái trước trong cái h<t-1>
- Tại bước thời gian t, ta có đầu vào mới là x<t>
- Thì lúc này cái x<t> sẽ được nhân mtran weight và rồi + thêm cái h<t-1> để ra trạng thái hiện tại.

  → Nói cách khác, trạng thái hiện tại được tính từ đầu vào hiện tại và trạng thái trong quá khứ. Vì cứ như vậy nên h<t-1> luôn luôn lưu thông tin của tất cả các thứ trước nó.

Vì sao RNN bị vanishing gradient?

Như trên ta đã có công thức cho RNN, ta biết 3 thứ cần train và ảnh hưởng đến kết quả là 3 cái W. Khi backpropagation, đầu tiên ta cần tính đạo hàm của loss theo 3 cái biến đó xong chain rule và quay ngược lại dần dần.

$$\frac{\partial L_t}{\partial W_{hh}} = \frac{\partial L_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_1}{\partial W_{hh}}$$

Mà mỗi cái h<t> theo công thức bên trên, ta có:

$$h^{<t>} = g_1(W_{hx} \cdot x^{<t>} + W_{hh} \cdot h^{<t-1>} + b_h)$$

Nên đạo hàm của nó theo h<t-1> sẽ là (g1 là hàm tanh):

$$\frac{\partial h_t}{\partial h_{t-1}} = \tanh'(W_{hh}h_{t-1} + W_{xh}x_t + b_h) \cdot W_{hh}$$

Thay ngược vào cái công thức bên trên và biểu diễn nó bằng ký hiệu dạng tích:

$$\frac{\partial L_t}{\partial W_{hh}} = \frac{\partial L_t}{\partial h_t} \cdot \prod_{t=2}^{T} \tanh'(W_{hh}h_{t-1} + W_{xh}x_t) W_{hh}^{T-1} \cdot \frac{\partial h_1}{\partial W_{hh}}$$

(1)

Chỗ đạo hàm này nếu mình k nhầm thì có thể suy ra từ

$$\frac{\partial h_t}{\partial h_{t-1}} = \frac{\partial h_t}{\partial z_t} \cdot \frac{\partial z_t}{\partial h_{t-1}}$$

- Với zt = Whh * h<t-1> + Wxh * xt
- ht = tanh(zt) nên đạo hàm của ht theo zt = tanh'(zt)
- Đạo hàm của zt theo h<t-1>, ta coi h<t-1> là biến còn lại là hằng số, thì cũng chỉ còn Whh.
  => Như vậy, sau mỗi bước lại nhân thêm Whh vào nên mới có cái Whh mũ t-1 bên trên.

Cái ***(1)*** này trông hơi lằng nhằng nhưng nhìn chung nó chỉ là nhân một đống vào thôi, cứ bước sau khi lại nhân với cái h bước trước.

- Vấn đề bắt đầu xuất hiện ở chỗ này: ta đều biết hàm tanh có đồ thị đạo hàm cũng khá bão hòa ở 2 đầu và giá trị max là 1, khi ***x rất âm hoặc rất dương, đạo hàm của hàm tanh sẽ tiến dần về 0.***

Với kiểu nhân chuỗi thế này, ta hình dung đơn giản nếu mỗi phần tử của cái tích này nhỏ hơn 1 (thật ra nó chắc chắn là nhỏ hơn 1 vì max của tanh' chỉ bằng 1 thôi), thì cái tích càng ra xa nó sẽ càng giảm và nhỏ dần, đến cuối cùng là mất hút.

Nhưng có thể nó không giảm quá nhanh như vậy, mấu chốt của việc này còn nằm ở một yếu tố khác là cái Whh.

- Nếu ***giá trị riêng của nó < 1***, gradient sẽ bị giảm một cách nhanh chóng sau vài bước, vanishing gradient luôn.
- Nếu ngược lại ***nếu giá trị riêng > 1***, thì gradient lại tăng theo cấp số nhân, thì lại exploding gradient.

Giá trị riêng thì ảnh hưởng gì ở đây?
Do đọc khó hiểu quá nên mình viết luôn cả đoạn giải thích mình đọc được ở đây, mặc dù chắc không ai đọc đâu nhưng nhỡ sau này mình đọc lại còn biết.

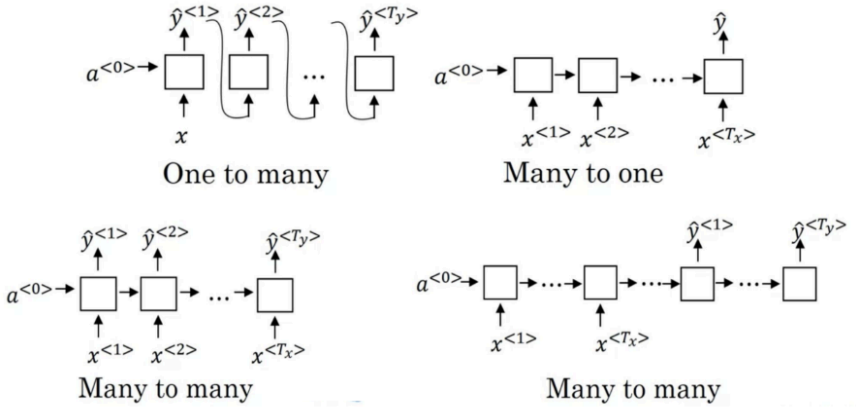
Cái này xuất phát từ Eigen Decomposition, ta có thể phân tích Whh thành (cái này chỉ áp dụng với ma trận vuông):

$$W_{hh} = P\Lambda P^{-1}$$

Với P ma trận vector riêng được xếp từng cột vào, cái ở giữa là ma trận đường chéo chỉ chứa các giá trị riêng. Mà thật ra vector riêng cũng phần nào được tính ra từ giá trị riêng.

Thì giờ nếu giá trị riêng nhỏ hơn 1, cái Whh còn phải mũ lên t-1 với t là số bước, bạn hình dung 1 đống số nhỏ hơn 1 nhân vào nhau, nhân thêm với cái tanh' đã nhỏ hơn 1 sẵn → kiểu domino nhỏ dần.

Một số type RNN:

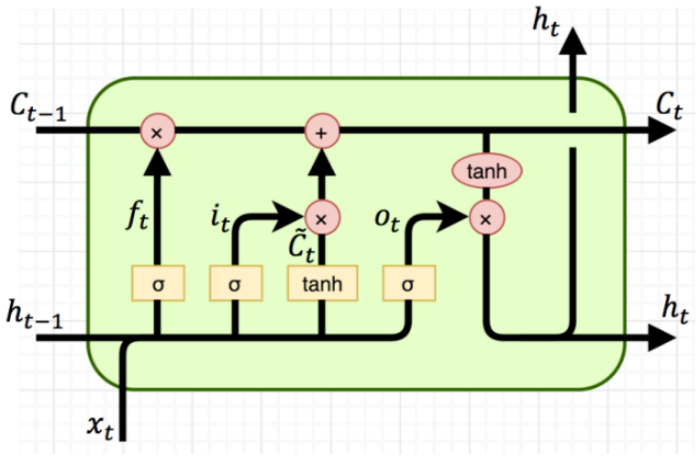


2. *LSTM (Long-Short Term Memory)*

Như ta thấy ở RNN, thông tin càng ra sa thì gradient nó càng biến mất, vừa gây k học được gì cũng như không có x gì. LSTM bổ sung thêm 1 vài thứ để khắc phục điều đó, cụ thể nó giữ những gì quan trọng trong chuỗi dài và quên mấy cái linh tinh k quan trọng.

- Hoa mỹ là thế chứ thật ra về mặt con số nó là về độ lớn của trọng số. Cái này sau khi viết công thức sẽ nói rõ hơn.

- Dưới đây là hình minh họa cách hoạt động của LSTM (ảnh lấy từ medium):



Viết theo kiểu toán học:

$$f_t = \sigma(W_f x_t + U_f h_{t-1})$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1})$$

$$\tilde{C}_t = \tanh(W_c x_t + U_c h_{t-1})$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1})$$

$$h_t = o_t \odot \tanh(C_t)$$

Giải thích nhanh: ft, it, ot chính là các cổng, đây là các khái niệm quan trọng trọng trong LSTM. ***Trong kiến trúc LSTM có 3 cổng:***

- ***Forget Gate (ft):*** xem giữ lại bao nhiêu thông tin từ trạng thái trước. Hàm sigmoid convert đầu ra về 0, 1 giống như còn nhớ bao nhiêu %, kiểu z.
- ***Input Gate (it):*** Dùng bao nhiêu thông tin từ đầu vào hiện tại. xt chính là input của bước thời gian t như đã nói trên RNN.
- Cái Ct~ y hệt như RNN.
- ***Output Gate (ot):*** Xem dùng bao nhiêu thông tin làm đầu ra.

Sự cải tiến lớn của LSTM so với RNN chính là cái Ct kia (Cell state), nó kiểu lưu trữ thông tin chạy xuyên suốt cái chuỗi. Nói chung nó không có gì quá phải giải thích. Nó gọi là cell state vì nó chỉ dùng để "trung chuyển thông tin" nội bộ trong đoạn giữa của mỗi step. Hiểu một cách nôm na là thế.

- Ý nghĩa của cái từ Long Term Memory cũng là từ cái cell này, nó kiểu sẽ mang đồ đi sang các bước sau, cái RNN bị gọi là short term vì mấy cái state ở xa nó không học được gì còn bị vanish.

Và chắc chắn rồi, LSTM nhiều thứ cần học hơn (cụ thể là phải học hết 3 cặp W U cho 3 cổng) và công thức phức tạp hơn, nên nó sẽ ***chậm hơn RNN.***

Vì sao LSTM khắc phục vanishing gradient?

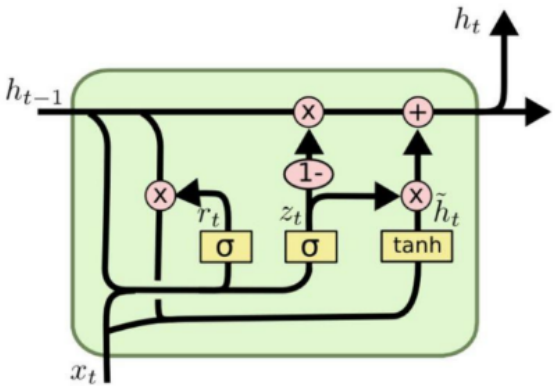
Chính là nhờ cái Cell State này. Bạn để ý nó được truyền thẳng sang bước thời gian tiếp theo mà không phải trải qua những phép nhân liên tiếp. Điều này khác hoàn toàn so với hidden state ở RNN khi nó phải nhân 1 đống tanh' các thứ. Để giải thích kỹ hơn thật ra phải đưa vào công thức đạo hàm, nhưng nó hơi dài và khó hiểu nên mình chưa ngấm hết.

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

Nhưng một cách đơn giản ta có thể thấy Ct được cập nhật bằng phép cộng, sẽ đỡ hơn như bên RNN nhân 1 đống.

3. *GRU (Gated Recurrent Unit)*

Ngoài ra còn 1 cái gọi là GRU, nó cũng có cơ chế kiểu kiểu LSTM, thật ra nó là một biến thể đơn giản hóa của LSTM, thay vì có 3 cổng thì nó còn có 2 cổng thôi:



(b) Gated Recurrent Unit

2 cổng đó là ***Reset Gate*** và ***Update Gate***, dưới đây là công thức của GRU:

2 cổng: rt (reset gate), zt (update gate):

$$r_t = \sigma(W_r \cdot h_{t-1} + U_r \cdot x_t + b_r)$$

$$z_t = \sigma(W_z \cdot h_{t-1} + U_z \cdot x_t + b_z)$$

Trạng thái tiềm năng mới:

$$\tilde{h}_t = \tanh(W \cdot (r_t \odot h_{t-1}) + U \cdot x_t + b)$$

Trạng thái cuối cùng:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

Cái này phần nào cũng khắc phục được Vanishing Gradient, đặc biệt nó ít tính toán hơn nên nhanh hơn LSTM là rõ, tất nhiên trong thực tế vẫn là ***try to find the best combination***, nhưng ta có thể đoán nó xử lý long context không tốt bằng LSTM, nhưng chắc chắn sẽ nhẹ và nhanh hơn.